Evaluating Non-Adequate Test-Case Reduction

Omitted for Double-Blind Review

ABSTRACT

Given two test cases, one longer and one shorter, the shorter test case is preferred for many purposes. A shorter test case usually runs faster, is easier to understand, and is more convenient for debugging. However, shorter test cases also tend to cover less code and detect fewer faults than longer test cases. While traditional work focused on reducing test suites while preserving code coverage, one line of recent work has introduced the idea of reducing individual test cases, rather than test suites, while still preserving code coverage; and another line of recent work has proposed non-adequately reducing test cases. This paper empirically evaluates a new combination of these ideas: non-adequate reduction of test cases, which allows for a wide range of trade-offs between test case size and fault detection.

Our study introduces and evaluates C%-coverage reduction (where a test case is reduced to retain at least C% of its original coverage) and N-mutant reduction (where a test case is reduced to kill at least N of the mutants it originally killed). We evaluate the reduction trade-offs with varying values of C and N for four real-world C projects, using both manually written and automatically generated test cases: Mozilla’s SpiderMonkey JavaScript engine, the YAFFS2 file system, Grep, and Gzip. The results show that it is possible to substantially reduce the size of many test cases while still preserving much of their fault-detection capability.

1. INTRODUCTION

Smaller test cases are, in many ways, preferable to longer test cases. For example, smaller test cases tend to run faster, which can improve the efficiency of test suites [7]. Smaller, simpler test cases are also easier to understand and enable effective debugging. In fact, this provided the original motivation for delta-debugging [24], a technique for reducing the size of failing test cases. The advantages of smaller test cases are so evident that random test generation is often combined with delta-debugging, and the development of effective reduction techniques is itself an important research area [7, 9, 14, 16]. Finally, smaller test cases make it possible for test case selection and prioritization to operate at a much finer granularity than when applied to test suites mostly consisting of large, complex test cases.

However, it is also true that smaller test cases, all else being equal, detect fewer faults than larger test cases [3]. The trade-off between size and effectiveness for individual test cases is similar to the trade-off between benefits of smaller and larger test suites that are sets of individual test cases. Researchers have extensively studied test-suite reduction [10, 11, 17, 19, 23], which removes entire test cases from test suites.

The problem of test-suite reduction is to reduce a given test suite while preserving most of its fault-detection capability. Various techniques have been proposed, many of which are summarized in a survey by Yoo and Harman [23]. Test-suite reduction trades-off reduction in the test-suite size (typically measured by the number of test cases) for reduction in the fault-detection capability (most often measured by the number of killed mutants). Most of these techniques completely preserve some property of a test suite, e.g., its code coverage, while removing test cases that are redundant and do not contribute to that property. However, recent work on non-adequate test-suite reduction [19] evaluated only partially preserving some property, e.g., keeping 90% of code coverage.

The problem of test-case reduction is to reduce an individual test case while preserving most of its fault-detection capability. Reducing a test case requires “slicing and dicing” the atomic parts that make the test case. For example, if a test case is a unit test comprising a sequence of function calls, reduction usually involves removing function calls. If a test case is defined by an input file, reduction can involve removing lines or characters from the file. Note that measuring the size of a test case is inherently project-specific and depends on the semantics of test cases, whereas the size of test suites can be defined in a project-agnostic way (though not perfectly correlated with the test execution time) as the number of test cases in the test suite. While test-suite reduction has been studied in depth at least since 1993 [11], test-case reduction research is much more recent.

Zeller proposed delta-debugging [24], the best known test-
case reduction technique, usually applied to reduce a failing test case to a minimal test case that still fails. Recently, Groce et al. proposed cause reduction [7] to reduce a (passing or failing) test case, while preserving coverage obtained by the original test case. Cause reduction completely preserves coverage: the reduced test case has exactly the same coverage as the original test case. We refer to such reductions as adequate because they preserve 100% of some property. Note that “adequate” in our context refers to the relationship between the reduced and original test cases, but the original test case itself may provide far from adequate code coverage.

[we should cite your STVR’16 paper to avoid any potential double submission or plagiarism issues]

The utility of non-adequate reduction for test suites [19] naturally suggests that non-adequate reduction may be useful for test cases as well. Non-adequate reduction for suites or test cases greatly enlarges the number of points to explore in trading off size and fault-detection capability. For test cases, adequacy limits how much reduction can be obtained, and increases the time required to reduce test cases. [Darko doesn’t understand the previous sentence] [grep (-i) to the rest (if not already there), and remove this highlight to the rest] Combining the recent ideas of non-adequate reduction for test suites [19] and (adequate) reduction for test cases [7], this paper empirically evaluates a new combination: non-adequate reduction of test cases. To the best of our knowledge, ours is the first such evaluation.

Specifically, we evaluate C%-coverage reduction (where a test case is reduced to retain at least C% of its original coverage) and N-mutant reduction (where a test case is reduced to kill at least N of the mutants it originally killed). Both reduce a larger test case to a smaller test case while only partially preserving some property. Hence, we refer to these reductions as “non-adequate” because they do not necessarily preserve completely either the code coverage or all mutants killed. However, the reduced test cases could, in theory cover code elements or kill mutants that the original test cases do not, even if the reduced test cases do not cover all code elements or kill all mutants that the original test cases do; in fact, the reduced test cases can even cover more code or kill more mutants overall.

These non-adequate test-case reduction further generalize previously proposed test-case reductions. By parameterizing the level to which a property needs to be preserved in the reduced test case, we allow more freedom to explore trade-offs between (more) size reductions and (less) preservation of the fault-detection capability [19,22]. For example, cause reduction [7] becomes just a special case of our C%-coverage when we set C = 100, and preserving to kill only one mutant that encodes some fault (N-mutant with N = 1) can mimic delta-debugging. At the other extreme of the spectrum, setting N in our N-mutant to equal the total number of all mutants killed by the original test case results in a very strict test-case reduction that requires preserving all mutants killed. Of course, such reduction may be prohibitively expensive to perform (and would likely provide very little reduction unless test cases have excessive redundancy), so our evaluation largely concerns only small values for N.

We evaluate non-adequate test-case reduction on four small-to-medium, real-world, important C projects—Mozilla’s SpiderMonkey JavaScript engine, the YAFFS2 flash [please please please put the four projects in the same order through-out! someone should grab a global lock and ensure this is the case; right now we have a difference even between intro and abstract] file system, Grep, and Gzip—some with manually written and some with automatically generated test cases. We evaluate C%-coverage for various levels of C%, from 70% to 100%, in increments of 10%. We also evaluate N-mutant with (1) randomly selected mutants for various values of N, from 1 to 32, in powers of 2, and (2) with carefully selected mutants that are hard to kill, based on the minimal mutant set [7] and [unique] mutants. We measure size reduction and, as a proxy for fault-detection capability, code coverage and mutants killed. Our results show that in many cases, non-adequate test-case reduction can substantially reduce given test cases size while still preserving considerable fault-detection capability. The results also show that C%-coverage and N-mutant provide a statistically significant better trade-off than random test-case reduction, when controlled for the size reduction.

[From Darko: the two contributions look fine now, but if we need to save space, we may just sprinkle the text from here to the rest (if not already there), and remove this highlight of two contributions. The highlight may make them appear rather small.]

This paper makes the following contributions:

- **Novel test-case reduction approach:** We introduce non-adequate test-case reduction via C%-coverage and N-mutant, which reduce the size of test cases while partially preserving some properties of the original test cases.

- **Evaluation of reduction trade-offs:** We evaluate, over four real-world C projects, the size reduction obtained with varying parameters, and the code coverage and mutation score for reduced test cases relative to the original, unreduced test cases.

2. Non-Adequate Test-Case Reduction

In this section, we precisely define the two non-adequate test-case reductions evaluated in this paper. Section 2.1 introduces some common notation. Section 2.2 describes the existing cause reduction algorithm. Section 2.3 presents C%-coverage, and Section 2.4 presents N-mutant.

### 2.1 Preliminaries

We introduce some notation that allows precisely specifying our test-case reductions. We use t to denote an arbitrary test case, Cov(t) to denote the set of statements covered by t, Mut(t) to denote the set of mutants killed by t, |S| to denote the cardinality of the set S, and Size(t) to denote the size of t. Measuring the size of a test case is specific to the project or test case format; Section 4.1.1 describes in detail how we define size for the projects used in our evaluation. Conceptually, we define size as the number of atomic parts that a test case has. We use tv to denote the original test case and τr to denote the reduced test case. The parameterized nature of parts is carried over from the original delta-debugging work [24], where a definition of test case [components – why are we using “components” here rather than “parts”? should that macro expand to “components”?]

---

[3] While we present and evaluate C%-coverage only for statement coverage, it generalizes to any other coverage, e.g., branch coverage.
was an input to the algorithm. In some applications, parts are function calls; in other applications they are lines or characters in a file; and in rarer applications they may be much more complex, defined by a grammar or another input specification. For example, reduction of test cases that are themselves computer programs (e.g., and input to a compiler or a static analysis tool) [16] often relies on a semantically involved and even variable notion of part that defies easy description.

The high-level goal of test-case reduction is to produce a reduced test case \( t_r \), with size smaller than the size of \( t_o \), i.e., \( \text{Size}(t_r) < \text{Size}(t_o) \) (and ideally \( \text{Size}(t_r) \ll \text{Size}(t_o) \)), such that \( t_r \) still retains (either completely or partially) some desirable property of \( t_o \). That is, for some notion of quality, \( t_r \) has similar “quality” to \( t_o \). \( t_o \) itself, of course, may have poor quality, but \( t_r \) should not have much worse quality than \( t_o \). While in principle the reduction process can stop at various steps (and in the limit, even the original test case can be considered a reduced version of itself), we are interested in so called “1-minimal” test cases [24] where no single part of \( t_o \) can be removed without losing the desired property of \( t_o \).

### 2.2 Reduction Algorithm

The test-case reduction algorithm we use is derived from the original delta-debugging [24] algorithm, but we modify it to support non-adequate test-case reduction. Delta-debugging reduces a failing test case by removing behaviors that are not relevant to the fact that the test case fails.

A generalized algorithm for cause reduction [7, 8] extends delta-debugging to reduce a test case with respect to any behavior; not just failure, that can be detected when running the test case.

**Algorithm 1** \( \text{ddmin} \) algorithm for simplification of failing test case, \( \text{ddmin}(P, C_X, n) \)

**Input:** \( P \), Program, \( C_X \), Failure-inducing Input, and \( n \), number of partitions.

Split \( C_X \) to \( n \) partitions to build \( c_1, \ldots, c_n \).

**if** \( \exists c_i \) such that makes \( P \) fail **then**

\( \text{ddmin}(P, c_i) \)

**else if** \( \exists c_i \) such that \( C_X - c_i \) makes \( P \) fail **then**

\( \text{ddmin}(P, C_X - c_i, \max(n - 1, 2)) \)

**else if** \( n < \min(C_X) \) **then**

\( \text{ddmin}(P, C_X, \min(2n, \min(C_X))) \)

**else**

\( C_X \)

**end if**

At a high level, the algorithm iteratively splits the test case in half, and checks if the behavior remains in either half; if any half preserves the behavior, it is used for the next iteration. (In the best case, the algorithm can end up performing binary search to produce a test case of size 1.) If no half preserves the behavior, then the algorithm backtracks, (do we need to introduce any other word but “backtrack” to refer in the text later?) “mixes and matches” the parts from the halves to search for a test case that preserves the behavior. The algorithm continues this loop until it produces a “1-minimal” test case.

[from DARK: do we still need this paragraph? If no, delete all this. If yes, we need to improve the wording. I don’t understand what exactly we’re trying to say here, and we introduce “ddmin” out of the blue.] Groce et al. used the \( \text{ddmin} \) algorithm size reduction of test cases, while preserving the coverage. We apply a further modification to this generalized cause reduction algorithm, relaxing the constraint of adequacy to allow for less than adequate test cases.

### 2.3 C%-coverage Reduction

Suppose that we are given a test case \( t_o \) that covers some set \( \text{Cov}(t_o) \) of statements in the program under test. Previously proposed cause reduction [7] produces a reduced test case that still covers all of \( \text{Cov}(t_o) \) (and could potentially cover even more statements). However, cause reduction of large test cases for complex software can be highly inefficient, because it involves searching many test cases, and for each test case computing coverage. For example, Groce et al. [7] reported that cause reduction of a large test case for the GCC compiler could take days. Moreover, preserving 100% of the coverage may not be necessary, because a test case that preserves less may still have high quality.

\( \text{C} \%-\text{coverage} \) relaxes the coverage requirement; the reduced test case need not necessarily preserve 100\% but should preserve at least \( \text{C} \% \) of the statements covered by \( t_o \):

**Definition 1.** \( \text{C}\%-\text{coverage} \) test-case reduction produces a reduced test case \( t_r \), that covers at least \( \text{C}\% \) of the statements that are covered by the original test case \( t_o \):

\[
\frac{|\text{Cov}(t_r) \cap \text{Cov}(t_o)|}{|\text{Cov}(t_o)|} \geq \text{C}\%.
\]

Note that the percentage is bound by the coverage of the original test case and not bound by all the statements in entire program under test, i.e., we do not want simply \( |\text{Cov}(t_r)| / |\text{Cov}(t_o)| \geq \text{C}\% \), because \( t_r \) could then be covering statements not related to \( t_o \). Viewed this way, cause reduction can be (re)defined as requiring \( |\text{Cov}(t_r) \cap \text{Cov}(t_o)| / |\text{Cov}(t_o)| = 100\% \), or equivalently \( \text{Cov}(t_r) \supseteq \text{Cov}(t_o) \).

In other words, \( \text{C}\%-\text{coverage} \) does not put any (direct) requirement for \( |\text{Cov}(t_r)| \) and \( |\text{Cov}(t_o)| \), so it may be even the case that \( |\text{Cov}(t_r)| > |\text{Cov}(t_o)| \) if \( t_r \) covers some statements that \( t_o \) does not cover. In other words, \( \text{C}\%-\text{coverage} \) does not impose any requirement on the statements that are not covered by the original test case: the reduced test case may or may not cover those statements.

### 2.4 N-mutant Reduction

We define \( N \)-mutant reduction similarly to \( \text{C}\%-\text{coverage} \) reduction, but with three differences: (1) we use killed mutants instead of covered statements; (2) we preserve a set of \( N \) selected mutants rather than a relative ratio of mutants; and (3) we use the same set of selected mutants for all reduction steps.

The difference (2) is motivated by the following reasoning. Suppose that we are given a test case \( t_o \) that kills some mutants \( M(t_o) \) in the program under test. We could require that the reduced test case \( t_r \) kills all those mutants. However, searching for such a reduced test case would be extremely inefficient because each step of reduction would require to check that the intermediate test cases kill all the mutants. Also, it is likely unnecessary to preserve all the mutants; the developers may be interested in some specific mutant or a handful of “hard to kill” mutants. Reducing the test case to preserve a small set of mutants can still produce a highly useful smaller test case, and reducing test cases
based on a limited number of mutants is definitely much more efficient than reducing with respect to the entire set of mutants.

\( N \)-mutant requires the reduced test case to preserve some specific \( N \) mutants selected from the set of all mutants killed by the original test case:

**Definition 2.** \( N \)-mutant test-case reduction produces a reduced test case \( t_r \) that has to kill a specific set of \( N \) mutants selected from the set of \( \text{Mut}(t_o) \), where typically \( N \ll |\text{Mut}(t_o)| \).

Note that we only require the selected \( N \) mutants to be a subset of \( |\text{Mut}(t_o)| \). Mutants other than those in the selected \( N \) mutants may or may not be killed by the test case \( t_r \).

The difference (3) is motivated by the following reasoning. Unlike for \( C\% \)-coverage, which does not keep the set of statements constant among reduction steps but only requires that a certain number of those statements be covered, \( N \)-mutant does keep the set of \( N \) selected mutants constant. We did initially experiment with allowing the set of mutants to change, while requiring only that the number of mutants be preserved through reduction steps be at least \( N \). However, by allowing the algorithm to only preserve at least any \( N \) mutants, it can be necessary to run a large number of mutants at each step of the reduction algorithm (until at least \( N \) mutants are killed or all mutants are run and \( N \) are not killed). As such, the time to perform non-adequate test-case reduction that preserves at least any \( N \) mutants was often prohibitively long. [let’s just mention speed here for (3) if we can’t easily talk about the quality]

## 3. METRICS

In this section, we describe three metrics for evaluating the effectiveness of test-case reduction: Size Reduction Rate (SRR), Coverage Preservation Rate (CPR), and Mutation Preservation Rate (MPR). We make all metrics such that the higher values are better and that the values are normalized to range between 0 and 1. We also relate the metrics with our techniques.

### 3.1 Size Reduction Rate (SRR)

The goal of test-case reduction is to reduce the size of a test case. As such, a key measure of quality of test-case reduction is how much smaller the reduced test case is compared to the original test case. Recall that \( \text{Size}(t) \) denotes the size of a test case \( t \), i.e., the number of the atomic parts that the test case has.

**Definition 3.** For an original test case \( t_o \) and its reduced test case \( t_r \), Size Reduction Rate (SRR) measures the reduction in size of the reduced test case relative to the original test case:

\[
\text{SRR}(t_o, t_r) = \frac{\text{Size}(t_o) - \text{Size}(t_r)}{\text{Size}(t_o)}
\]

A higher ratio for SRR is desirable because it indicates that more parts have been removed from the test case, resulting in a smaller reduced test case.

### 3.2 Coverage Preservation Rate (CPR)

Our reduction is non-adequate test-case reduction, so we need some metrics to measure how much fault-detection capability the reduced test case loses compared to the original test case. Structural code coverage is commonly used as a proxy for fault-detection capability to evaluate the quality of test cases; the more code a test case covers the higher chance it can detect a fault, and conversely if a test case fails to cover some part of code there is no way it can detect any faults in that part of the code.

An number of studies have shown that code coverage is a strong indicator of fault detection capability [5, 6]. We note that while the study by Inozemtseva et al. [12] seems to suggest otherwise, we note that Inozemtseva et al. looked at the effect of coverage after discounting the effect of size of test suite. If the effect of test suite is not controlled for, Inozemtseva et al. [12] found a strong correlation between code coverage and fault detection capability of a test suite. We use statement coverage as the structural code coverage metric to evaluate quality. Recall that \( \text{Cov}(t) \) denotes the set of statements covered by a test case \( t \).

**Definition 4.** For an original test case \( t_o \) and its reduced test case \( t_r \), Coverage Preservation Rate (CPR) measures the ratio of the number of statements covered by the reduced test case that are also covered by the original test case, to the number of statements covered by the original test case:

\[
\text{CPR}(t_o, t_r) = \frac{|\text{Cov}(t_r) \cap \text{Cov}(t_o)|}{|\text{Cov}(t_o)|}
\]

A higher ratio for CPR is desirable because it indicates the reduced test case covers a larger portion of statements covered by the original test case. Note that a reduced test case can potentially cover more statements than the original test case, but CPR only considers the statements covered by the original test case, and as such the ratio can never go over 1.

### 3.3 Mutation Preservation Rate (MPR)

Another metric commonly used to evaluate the quality of test cases is the number of killed mutants. We measure how effective the reduced test case is at killing mutants when compared to the original test case. Recall that \( \text{Mut}(t) \) denotes the set of mutants killed by a test case \( t \).

**Definition 5.** For an original test case \( t_o \) and its reduced test case \( t_r \), Mutation Preservation Rate (MPR) measures the preservation of mutants killed by the reduced test case with respect to the mutants killed by the original test case:

\[
\text{MPR}(t_o, t_r) = \frac{|\text{Mut}(t_r) \cap \text{Mut}(t_o)|}{|\text{Mut}(t_o)|}
\]

A higher MPR is desirable because it indicates the reduced test case is better at preserving the ability to kill mutants that the original test case kills. Note that a reduced test case can potentially kill more mutants than the original test case, but MPR does not consider the other mutants that are killed by the reduced test case but not killed by the original test case. Like CPR, MPR can never go over 1.

### 3.4 Relating Metrics and Criteria for Reduction
Both the reduction algorithm and the metrics are based on coverage and mutants, but note that the requirements for reduction are not the same as the metrics used to evaluate the reduced test cases. Therefore, we cannot a priori tell how high or low the metrics will be for different reductions. For C%-coverage reduction, we know that CPR will be at least C% but it could be much higher (though always less than or equal to 1), and MPR could in theory range from literally 0 to 1. For N-mutant reduction, we know that MPR will be at least $N/|\text{Mut}(t_o)|$, but it could be much higher (in fact, our experiments find that even when $N/|\text{Mut}(t_o)| < 1\%$, MPR can be more than 50%), and CPR could again in theory range from literally 0 to 1.

4. EVALUATION

In this section, we describe how we designed the experiments to evaluate the effectiveness of our C%-coverage and N-mutant non-adequate test-case reduction techniques. Section 4.1 describes the projects we used in our evaluation, along with some characteristics about their test cases and the mutants generated for these projects. Section 4.2 describes our experimental setup.

We ran all our experiments on a high-performance computing cluster composed of commodity computing nodes. Each node had between 6 to 12 2.6Ghz Intel Xeon cores. All experiments together would take weeks on a single core, mostly due to the high cost of mutation testing.

4.1 Projects

Table 1 lists the projects we used in our evaluation. We tabulate the project name, its size, the number of test cases used in our evaluation, what the smallest delta part of each test case is, the total number of mutants, and the minimum and maximum number of mutants killed by each test case. We use four small- to medium-size C projects in our evaluation: SpiderMonkey is the Mozilla’s JavaScript engine, YAFFS2 is a flash file system that was used in the early Android platforms, Grep is the standard Unix utility for searching files, and Gzip is the standard Unix utility for compressing/decompressing files. These projects range from 5,129 to 81,920 non-comment lines of code.

4.1.1 Test Cases

We use automatically generated test cases for SpiderMonkey, YAFFS2, and Gzip, and we use manually written test cases for Grep. More precisely, for SpiderMonkey, YAFFS2, and Gzip, we randomly generate test cases for each project, as done in similar previous work [7]. The SpiderMonkey test cases are JavaScript programs randomly generated using a highly successful jsfunfuzz [18] fuzzier. The YAFFS2 test cases are sequences of API calls to the file system, randomly generated using a publicly available test generator for YAFFS2 that has been used by several research projects on test generation. The Gzip test cases are files that are randomly generated using files, and Gzip, for which we generate the test cases, we control the number of parts such that test-case reduction can finish in reasonable time for each test case. We obtained the specific limits from the initial experiments, trying to finish most test-case reductions within 30 minutes. In particular, we limit each SpiderMonkey test case to be a program consisting of exactly 200 JavaScript statements, each YAFFS2 test case to be a sequence consisting of exactly 200 API calls, and each Gzip test case to be a file consisting of at most 3,500 bytes. For Grep, we use all the test case manually written by others [4], and we do not limit their size; the longest test case for Grep consists of 146 characters in the command-line.

4.1.2 Mutants

We use a mutation-testing tool for C programs developed by Andrews et al. [2] and used in many previous studies. Quoting [2], the tool provides the following four classes of mutation operators:

- Replace an integer constant $I$ by $0$, $1$, $-1$, $((I) + 1)$, or $((I) - 1)$;
- Replace an arithmetic, relational, logical, bit-wise logical, increment/decrement, or arithmetic-assignment operator by another operator from the same class;
- Negate the decision in an if or while statement;
- Delete a statement.

The tool performs source-to-source mutations, so the mutants are source code. We compile each mutant with the highest optimization available in the GCC compiler (-O3) and further compare each binary with the others to keep only one representative from each equivalence class of mutants that are trivially compiler-equivalent [15]. (We note that around 15% of the generated mutants are considered equivalent mutants by use of trivial compiler equivalence analysis.) Table 1 shows the number of mutants for each project and the minimum and maximum number of mutants killed by each test case. A mutant is considered killed if its output differs from the output of original program; the output includes stdout and stderr.

4.1.3 Minimal Mutants

In order to choose a reasonable set of mutants for the N-mutant reduction, we looked at two methods. The first method was random sampling of $N$ mutants from the complete set of mutants. To contrast with this random sample, we also chose the minimal set of mutants for each project. Minimal set of mutants as defined by Ammann et al. [1] is
the smallest set of mutants that represents the complete set of mutants in terms of faults detected by a given test suite. A minimal mutant set is important in that it can be said to represent all the faults in a set of mutants that was exposed to a test suite, such that killing all mutants in the minimal mutant set guarantees killing all mutants in the complete set of mutants.

To define the minimal mutant set, the first step is to construct a minimal test suite from the original test suite. A minimal test suite is the smallest set of tests from the original test suite such that it is mutation adequate against the set of mutants killed by the original test suite. That is, the minimal test suite scores 100% against the mutants killed by the original test suite, and removing any of the test cases results in a drop in mutation score. Given a minimal test suite, a minimal mutant set is the smallest subset of mutants from the original mutant set such that the complete minimal test suite is required to kill all mutants in the minimal mutant set.

Our strategy was as follows: First, we generated random test cases to represent a test suite for each project. Note that these automatically generated test cases are different from the ones we run our non-adequate test-case reduction experiments on. Then, we applied the mutations on the projects, and obtained the complete pool of mutants that are killed by the test suite we generated for each project. For each pair of test suite and mutant set, we minimized the test suite with respect to the killed mutant set to obtain the minimal test suite. Using the minimal test suite, we minimized the mutant set to obtain the minimal mutant set for each project. The total number of test cases generated and the size of minimal mutant set are given in Table 2.

4.2 Experimental Setup

For C%-coverage test-case reduction, we perform experiments with the non-adequacy level C chosen from the set \{70,80,90,95,100\}. For each original test case, we create a reduced test case that preserves at least C% of the statements covered by the original test case, as required by Definition 1. We use GCov to obtain the set of statements covered by each test case.

For N-mutant test-case reduction, we perform experiments with the non-adequacy level N chosen from the set \{1,2,4,8,16,32\}. For each original test case, we first determine what mutants the test case kills and then randomly select N of those mutants. (For a small number of test cases that kill fewer than N mutants, we use all mutants.) We then create a reduced test case that preserves these N selected mutants, as required by Definition 2. To speed up the experiments, at each reduction step, we only run the N selected mutants (and not all the mutants killed by the original test case, let alone all the mutants generated for the project).

[Check later if it’s best to introduce this part here, and if everything I describe here is correct] In addition to randomly selecting N mutants for each test case, we also perform N-mutant test-case reduction by smartly selecting to preserve a subset of the mutants a test case kills during the reduction. We smartly select these mutants in two ways. First, we choose the mutants each test case kills that are in the minimal mutant set for the entire test suite. A property of minimal mutants is that each test can only kill at most 1 of the mutants in the minimal mutant set, so the value N here will always be 1. [Please check if that property is true...]. In the case of when a test case kills no mutants in the minimal mutant set, we do not reduce the test case and keep the test case as is. [Is this what we do, keep the original test case? Is this what we want?] [We don’t do the suite based reduction -Amin]

Performing non-adequate test-case reduction can take a long time for some test cases. We limit the reduction to 30 minutes per each test case. We observed that N-mutant test-case reduction starts having many timeouts when N gets greater than about 40, so we restrict our choices of N to values less than 40. We do not include not-reduced test cases in the results, because their CPR and MPR are know beforehand.

For each reduced test case, we further generate three randomly reduced test cases that have exactly the same size as the reduced test case. We create such a randomly reduced test case by starting from the original test case and iteratively choosing to remove (uniformly randomly selected) one by one part until the resulting test case has the same number of parts as the reduced test case. We perform random test-case reduction merely as a baseline for comparison; we do not necessarily recommend using random test-case reduction as a means to create smaller test cases because our analysis shows that such randomly reduced test cases have lower quality than test cases reduced by our C%-coverage or N-mutant techniques.

For each reduced test case, we finally measure the three metrics of SRR, CPR, and MPR. For SRR, we measure the number of parts in the reduced test case and compare it with the number of parts in the original test case, as per Definition 3. For CPR, we run both the original test case and the reduced test case using GCov to obtain how many statements are covered, and then use Definition 4. Similarly, for MPR, we run both the original test case and the

---

### Table 1: Four projects used in our evaluation and some statistics of their test cases and mutants

<table>
<thead>
<tr>
<th>Project</th>
<th>NCLOC</th>
<th># test cases</th>
<th>part</th>
<th># mutants</th>
<th>min killed</th>
<th>max killed</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpiderMonkey</td>
<td>81,920</td>
<td>99</td>
<td>A statement of JavaScript program</td>
<td>69,067</td>
<td>8101</td>
<td>12825</td>
</tr>
<tr>
<td>YAFFS2</td>
<td>10,356</td>
<td>99</td>
<td>One API call</td>
<td>15,046</td>
<td>2071</td>
<td>3439</td>
</tr>
<tr>
<td>Grep</td>
<td>8,433</td>
<td>112</td>
<td>A character in command-line [parameters]</td>
<td>7,591</td>
<td>19</td>
<td>993</td>
</tr>
<tr>
<td>Gzip</td>
<td>5,129</td>
<td>73</td>
<td>A byte in the input file</td>
<td>7,175</td>
<td>1813</td>
<td>2046</td>
</tr>
</tbody>
</table>

---

### Table 2: Statistics about minimal mutants computed for four projects

<table>
<thead>
<tr>
<th>Project</th>
<th># test cases</th>
<th># min-mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpiderMonkey</td>
<td>850</td>
<td>256</td>
</tr>
<tr>
<td>YAFFS2</td>
<td>1000</td>
<td>57</td>
</tr>
<tr>
<td>Grep</td>
<td>840</td>
<td>99</td>
</tr>
<tr>
<td>Gzip</td>
<td>1000</td>
<td>32</td>
</tr>
</tbody>
</table>
5. RESEARCH QUESTIONS

In our evaluation, we address the following five research questions:

- **RQ1**: How much can test cases be reduced in non-adequate test-case reduction (while preserving C%-coverage or N-mutant)?
- **RQ2**: How much coverage is preserved in non-adequate test-case reduction?
- **RQ3**: What percentage of mutants are preserved in non-adequate test-case reduction?
- **RQ4**: How does CPR in non-adequate test-case reduction compare to CPR in random test reduction?
- **RQ5**: How does MPR in non-adequate test-case reduction compare to MPR in random test reduction?

The following sections provide answers to our research questions.

### 5.1 RQ1: Size Reduction Rate

Figure 1 summarizes the results for SRR on the reduced test cases obtained using C%-coverage reduction. Similarly, Figure 2 summarizes the results for SRR on the reduced test cases obtained using N-mutant reduction. For each subject and each level of C and N, we see the distribution of SRR in the boxplots.

From both tables, we see that both C%-coverage and N-mutant test-case reductions can substantially reduce the size of test cases. Across all projects and values for C, the median SRR for all test cases reduced using C%-coverage is greater than 50%, i.e., the size of reduced test case is less than half the size of the original test case. Similarly, across all projects and values for N, the median SRR for all test cases reduced using N-mutant is greater than 50%.

From Figure 1, we also see that SRR decreases as C increases for C%-coverage, as should be expected. But it is interesting that SRR for C = 100 (requiring the reduced test case to cover all statement covered by the original test case, as in cause reduction [7]) is quite low compared with SRR for the other values of C; in other words, once we relax the coverage to allow missing a small portion of the statements, SRR increases substantially. For example, for Gzip, the median SRR for C = 100 and C = 95 differ over 20pp\(^2\).

The same observations can be drawn for N-mutant from Figure 2: the more we relax the constraint of the number of mutants that must be preserved during test-case reduction, the higher SRR gets as more and more parts can be removed from the original test case.

In sum, our answer for RQ1 is the following:

**RQ1**: We find that test cases can be reduced in size (as counted by the number of parts in a test case) by 50% or more, on average, using either of our non-adequate test-case reduction techniques. We further confirm that the more relaxed non-adequacy is, the greater the size reduction gets.

### 5.2 RQ2: Coverage Preservation Rate

Figure 3 summarizes the results for CPR on the reduced test cases obtained using C%-coverage reduction. Similarly, Figure 4 summarizes the results for CPR on the reduced test cases obtained using N-mutant reduction. For each subject and each level of C and N, we see the distribution of CPR in the boxplots.

Figure 4 illustrates the relation between different values of N and CPR. It shows that CPR in N-mutant test-case

---

\(^2\)The "pp" metric (from "percentage points") is used to represent differences in values that are already expressed as percentages.
5.3 RQ3: Mutation Preservation Rate

Figure 5 summarizes the results for MPR on the reduced test cases obtained using C%-coverage reduction. Similarly, Figure 6 summarizes the results for MPR on the reduced test cases obtained using N-mutant reduction. For each subject and each level of C and N, we see the distribution of MPR in the boxplots.

From the MPR results of C%-coverage reduction, we see that the range of the median MPR across all projects and all values of C is 39.48% to 93.19%. Concerning general trends, we see that MPR is positively correlated to the value of C, i.e., the bigger that C gets (resulting in a stricter constraint on the statements that need to be preserved by the reduced test case), the more mutants the reduced test case is able to kill relative to the mutants killed by the original test case. We see that at C of 95 or higher, the reduced test cases have the median at least 70% MPR for all projects. From conducting a Kendall-τ test, we see that there is a strong positive correlation between the value of C and MPR, as shown in Table 5.

Concerning trade-offs between SRR and MPR, Figure 20 shows the correlation between SRR and MPR, where each

### Table 3: Kendall test of association between N and MPR in N-mutant test-case reduction. All values significant at p < 0.01

<table>
<thead>
<tr>
<th>Project</th>
<th>Kendall τ_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpiderMonkey</td>
<td>0.66</td>
</tr>
<tr>
<td>YAFFS2</td>
<td>0.69</td>
</tr>
<tr>
<td>Grep</td>
<td>0.56</td>
</tr>
<tr>
<td>Gzip</td>
<td>0.51</td>
</tr>
</tbody>
</table>
on the minimal mutants they kill result in better trade-offs between SRR and MPR than for test cases reduced based on a random mutant it kills. By better, we mean that for the same SRR, test cases reduced based on minimal mutants tend to have a higher MPR than test cases reduced based on the same SRR test cases reduced based on minimal mutants tend to have a higher SRR as well.

<table>
<thead>
<tr>
<th></th>
<th>SpiderMonkey</th>
<th>YAFFS2 90</th>
<th>Gzip 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>1.00</td>
<td>3.28</td>
<td>3.28</td>
</tr>
<tr>
<td>Max</td>
<td>1.00</td>
<td>10.00</td>
<td>10.00</td>
</tr>
<tr>
<td>Mean</td>
<td>1.00</td>
<td>3.92</td>
<td>3.92</td>
</tr>
<tr>
<td>Median</td>
<td>1.00</td>
<td>4.00</td>
<td>4.00</td>
</tr>
<tr>
<td>SRR</td>
<td>10.00</td>
<td>10.00</td>
<td>10.00</td>
</tr>
<tr>
<td>MPR</td>
<td>80.00</td>
<td>80.00</td>
<td>80.00</td>
</tr>
</tbody>
</table>

Table 4: Time in seconds to perform C%-coverage test-case reduction.

When \( C = 100 \) in the C%-coverage reduction, our technique is the same as cause reduction, which was evaluated previously [7] (but not using a metric like MPR). We see that the MPR for cause reduction is higher than for other values of \( C \) for each project from our experiments. However, the time to perform cause reduction is also generally higher per test case. Table 4 shows the amount of time needed to perform C%-coverage reduction for each project and each value of \( C \). (For this metric, unlike the other metrics we use to evaluate reductions, the lower values are better as they indicate that the reduction took less time.)

From the MPR results of N-mutant reduction, we see that by focusing the reduced test case to preserve a small number of mutants killed by the original test case still kills a large fraction of the mutants killed by the original test case. For example, by reducing test cases based only on one mutant (i.e., \( N = 1 \)), the median MPR values are 60.26%, 31.39%, 80.28%, and 43.92% for SpiderMonkey, YAFFS2, Grep, and Gzip, respectively. These high MPR values for such a small \( N \) suggest that there are a lot of dependencies between mutants killed by a test case. Figure 12 illustrates this dependency. The x-axis shows the ratio of \( N \) to the total number of mutants killed by the original test case, i.e., \( N/Mut(t_o) \),

Figure 9: Relation between SRR and MPR – YAFFS2

Figure 10: Relation between SRR and MPR – Grep

Figure 8: Relation between SRR and MPR – SpiderMonkey

Figure 11: Relation between SRR and MPR – Gzip
and the y-axis shows the corresponding MPR. We can see that for SpiderMonkey, YAFFS2, and Gzip, a test case that is reduced based on less than 0.5% of the mutants killed by the original test case can still kill more than 50% of those mutants. [Is this true for Grep as well? It looks true...]

Comparing across the projects, we see that YAFFS2 has the lowest median MPR when reduced using N-mutant reduction (31.39%). YAFFS2 test cases are sequences of function calls to the file system API, such as `mount`, `open`, or `close`. There is little dependency across those functions (e.g., only a few functions call one another), so YAFFS2 test cases effectively control the interaction among the functions by the ordering of API calls. Thus, individual mutants can be isolated reasonably well from the other mutants, due to better decoupling between functions. On the other hand, modules in SpiderMonkey, like any other interpreter or compiler, are deeply intertwined to each other. That is, functions call one another to pass the results of lexing, parsing, interpretation, etc., and thus test cases have no means to affect that coupling. As a result, killing a mutant, say in the parsing module, may also correlate with killing many other mutants in passes before or after parsing, such as lexing or interpretation. Therefore, it seems reasonable that more than 75% of the reduced test cases based on a single mutant in SpiderMonkey could still kill more than 59% of the mutants killed by the original test cases.

Concerning trends, we find that as N grows, MPR of the reduced test cases increases, because more interdependent mutants can be killed. This observation is validated by the Kendall-τ test of association, shown in Table 3. It suggests that there is a strong positive association between N and MPR in the results. However, note that as N increases, the time to perform the reduction can increase as well, because the intermediate test cases need to be checked against more mutants, and the chance of timeout increases.

Also, the size of the original test case can affect the time to perform the reduction negatively.

In sum, our answer for RQ3 is the following:

**RQ3:** We find that test cases reduced using any non-adequate test-case reduction technique can on median preserve at least 31.39% of the mutants killed by the original test case. MPR grows as the non-adequacy (C or N) gets more strict, which we can confirm using a Kendall-τ test. Furthermore, we find that even when the N number of mutants to preserve in N-mutant test-case reduction is small relative to the number of all the mutants killed by the original test case, the reduced test cases can still kill a large percentage of the mutants killed by the original test cases.

### Table 5: Kendall test of association between C and MPR in C%-coverage test-case reduction. All values significant at p < 0.01

<table>
<thead>
<tr>
<th></th>
<th>Kendall ( \tau )</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpiderMonkey</td>
<td>0.89</td>
</tr>
<tr>
<td>YAFFS2</td>
<td>0.80</td>
</tr>
<tr>
<td>Grep</td>
<td>0.67</td>
</tr>
<tr>
<td>Gzip</td>
<td>0.76</td>
</tr>
</tbody>
</table>

5.4 **RQ4 & RQ5: Comparison with Random**
In this section, we compare our non-adequate test-case reduction techniques against the baseline of performing random test-case reduction. We run random test-case reduction on a number of test cases to compare the effectiveness with our non-adequate test-case reduction techniques. Because we perform random test-case reduction by matching the size of a reduced test case (as described in Section 4.2), SRR is exactly the same between a randomly reduced test case and its corresponding reduced test case made through a non-adequate test-case reduction technique. Therefore, we measure only CPR and MPR for these randomly reduced test cases. For each reduced test case generated by the non-adequate test-case reduction techniques, we generate three reduced test cases of exactly the same size, by randomly removing parts of the original test case.

### Table 6: Side-by-side comparison of CPR between C%-coverage test-case reduction and random test-case reduction.

<table>
<thead>
<tr>
<th></th>
<th>Min 25</th>
<th>50</th>
<th>mean</th>
<th>75</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-Coverage SpiderMonkey</td>
<td>0.65</td>
<td>0.80</td>
<td>0.90</td>
<td>0.96</td>
<td>1.00</td>
</tr>
<tr>
<td>Random SpiderMonkey</td>
<td>0.68</td>
<td>0.76</td>
<td>0.85</td>
<td>0.84</td>
<td>0.92</td>
</tr>
<tr>
<td>Random YAFFS2</td>
<td>0.32</td>
<td>0.39</td>
<td>0.45</td>
<td>0.44</td>
<td>0.48</td>
</tr>
<tr>
<td>C-Coverage Grep</td>
<td>0.70</td>
<td>0.85</td>
<td>0.97</td>
<td>0.92</td>
<td>0.99</td>
</tr>
<tr>
<td>Random Grep</td>
<td>0.25</td>
<td>0.61</td>
<td>0.84</td>
<td>0.77</td>
<td>0.94</td>
</tr>
<tr>
<td>C-Coverage Gzip</td>
<td>0.77</td>
<td>0.80</td>
<td>0.90</td>
<td>0.88</td>
<td>0.96</td>
</tr>
<tr>
<td>Random Gzip</td>
<td>0.63</td>
<td>0.63</td>
<td>0.64</td>
<td>0.68</td>
<td>0.77</td>
</tr>
</tbody>
</table>

Table 7: Side-by-side comparison of CPR between N-mutant test-case reduction and random test-case reduction.

<table>
<thead>
<tr>
<th></th>
<th>Min 25</th>
<th>50</th>
<th>mean</th>
<th>75</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-Mutant SpiderMonkey</td>
<td>0.89</td>
<td>0.95</td>
<td>0.98</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td>Random SpiderMonkey</td>
<td>0.67</td>
<td>0.73</td>
<td>0.76</td>
<td>0.77</td>
<td>0.80</td>
</tr>
<tr>
<td>N-Mutant YAFFS2</td>
<td>0.84</td>
<td>0.95</td>
<td>0.98</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td>Random YAFFS2</td>
<td>0.08</td>
<td>0.26</td>
<td>0.31</td>
<td>0.32</td>
<td>0.38</td>
</tr>
<tr>
<td>N-Mutant Grep</td>
<td>0.67</td>
<td>0.98</td>
<td>1.00</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td>Random Grep</td>
<td>0.12</td>
<td>0.61</td>
<td>0.84</td>
<td>0.76</td>
<td>0.94</td>
</tr>
<tr>
<td>N-Mutant Gzip</td>
<td>0.64</td>
<td>0.87</td>
<td>0.96</td>
<td>0.91</td>
<td>0.98</td>
</tr>
<tr>
<td>Random Gzip</td>
<td>0.63</td>
<td>0.64</td>
<td>0.78</td>
<td>0.78</td>
<td>0.92</td>
</tr>
</tbody>
</table>

Table 8: Side-by-side comparison of MPR between C%-coverage test-case reduction and random test-case reduction.

<table>
<thead>
<tr>
<th></th>
<th>Min 25</th>
<th>50</th>
<th>mean</th>
<th>75</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-Coverage SpiderMonkey</td>
<td>54.39</td>
<td>70.22</td>
<td>84.89</td>
<td>80.85</td>
<td>92.14</td>
</tr>
<tr>
<td>Random SpiderMonkey</td>
<td>53.76</td>
<td>65.10</td>
<td>75.78</td>
<td>76.22</td>
<td>85.25</td>
</tr>
<tr>
<td>Random YAFFS2</td>
<td>42.64</td>
<td>60.85</td>
<td>73.06</td>
<td>71.80</td>
<td>82.38</td>
</tr>
<tr>
<td>Random YAFFS2</td>
<td>0.60</td>
<td>1.94</td>
<td>2.44</td>
<td>4.44</td>
<td>3.14</td>
</tr>
<tr>
<td>C-Coverage Grep</td>
<td>27.47</td>
<td>82.69</td>
<td>89.30</td>
<td>87.90</td>
<td>96.54</td>
</tr>
<tr>
<td>Random Grep</td>
<td>1.32</td>
<td>47.65</td>
<td>63.63</td>
<td>64.04</td>
<td>81.74</td>
</tr>
<tr>
<td>C-Coverage Gzip</td>
<td>39.26</td>
<td>44.51</td>
<td>67.68</td>
<td>63.44</td>
<td>74.48</td>
</tr>
<tr>
<td>Random Gzip</td>
<td>35.79</td>
<td>27.19</td>
<td>28.14</td>
<td>32.75</td>
<td>40.86</td>
</tr>
</tbody>
</table>

Table 9: Side-by-side comparison of MPR between N-mutant test-case reduction and random test-case reduction.

We compared the CPR of randomly reduced test cases with test cases reduced using C%-coverage and N-mutant test-case reduction using non-parametric Wilcoxon Rank Sum test. We found that the sample of randomly reduced test cases are significantly different (p < 0.01) from the test cases reduced by a non-adequate test-case reduction technique. That is, the differences in the means we saw earlier in the side-by-side comparisons are likely true differences.

Table 6 shows a side-by-side comparison of CPR for test cases reduced by C%-coverage and test cases reduced randomly. Similarly, Table 7 shows a side-by-side comparison of CPR for test cases reduced by N-mutant and test cases reduced randomly. We see from these tables that the mean CPR computed for test cases reduced by a non-adequate test-case reduction technique is greater than the mean CPR computed for the test cases reduced by random test-case reduction. Tables 8 and 9 show the same comparison, except this time for MPR. Once again, we see from these tables that the mean MPR computed for test cases reduced by a non-adequate test-case reduction technique is greater than the mean MPR computed for the test cases reduced randomly.

### 6. DISCUSSION

Running mutation analysis of C programs on HPC cluster. Mutation analysis of C programs on shared computing resources can be challenging, because such experiments would include a variety of crashes that are continuously monitored in such systems and may trigger some actions by system administrators. We witnessed this problem in the course of running experiments for this study. The access to the HPC system was revoked multiple times because the system viewed the behavior of the experiments as suspicious. For example, rate of segmentation faults was monitored by the system, and some actions by the system viewed the behavior of the experiments as suspicious. Therefore, we performed random test-case reduction by matching the size of a reduced test case (as described in Section 4.2), SRR is exactly the same between a randomly reduced test case and its corresponding reduced test case made through a non-adequate test-case reduction technique. Therefore, we measure only CPR and MPR for these randomly reduced test cases. For each reduced test case generated by the non-adequate test-case reduction techniques, we generate three reduced test cases of exactly the same size, by randomly removing parts of the original test case.

In this section, we compare our non-adequate test-case reduction techniques against the baseline of performing random test-case reduction. We run random test-case reduction on a number of test cases to compare the effectiveness with our non-adequate test-case reduction techniques. Because we perform random test-case reduction by matching the size of a reduced test case (as described in Section 4.2), SRR is exactly the same between a randomly reduced test case and its corresponding reduced test case made through a non-adequate test-case reduction technique. Therefore, we measure only CPR and MPR for these randomly reduced test cases. For each reduced test case generated by the non-adequate test-case reduction techniques, we generate three reduced test cases of exactly the same size, by randomly removing parts of the original test case.

In sum, our answer for RQ4 and RQ5 is the following:

**RQ4 and RQ5:** We find that a test case that is randomly reduced to the same size as a test case reduced by a non-adequate test-case reduction technique is not as effective as its corresponding randomly reduced test case created using one of our non-adequate test-case reduction techniques. We find that for both CPR and MPR, the randomly reduced test case is not as good, and the difference is statistically significant according to a paired t-test.
that reducing with respect to a few random mutants preserves a large proportion of mutants killed. For example, in one experiment, preserving only nine mutants could preserve almost 98% of 2,500+ mutants in the original test cases. [should the previous word be test cases or test case?]

Note that reduction with respect to mutant X preserving mutant Y does not mean that X subsumes Y in the traditional sense where all tests detecting X detect Y. This poses several important research questions: (1) What is the extent of mutant domination in the programs? (2) Is there a smarter way than random selection to choose mutants? (3) Can we use the reduction data to cluster mutants?

Cascading reduction. N-mutant reduction decreases not only effectively reduces the size of the original test case, but the time to perform the reduction is shorter as well compared with trying to preserve all mutants killed by the original test case. However, N-mutant reduction does indeed sacrifice the ability for the reduced test case to kill some mutants. A remedy for this loss in mutant killing could be to run reduction multiple times to generate multiple test cases. That is, after a reduction ends, we start a new reduction on the reduced test case, but preserve in the N mutants the ones are not killed by the reduced test case, but were killed by the original test case. By continuously iterating until all mutants killed by the original test case are killed by the resulting reduced test case (essentially until MPR is 1), we can potentially create a reduced test case that is both smaller than the original test case yet kill the same mutants as the original, and this reduced test case could be made in a reasonable amount of time.

7. THREATS TO VALIDITY

Mutants of C programs can introduce undefined behavior. For example, the result of a mutant that removes initialization of a local variable from the program may vary, depending on the memory content of that variable. We did not inspect mutants for containing such behavior. However, we generate a large number of mutants for each of our subjects, so there is a smaller chance that mutants that introduce such undefined behaviors will significantly bias our results. [August: does the previous logic fly?]

[To add: Timeout in shared computing environment]

We measure the mutants killed after performing N-mutant test-case reduction even though N-mutant test-case reduction specifically uses what mutants are killed as the guidance to perform test-case reduction; one can say we are by construction making good reduced test cases that satisfy our metric. However, we are performing non-adequate test-case reduction, meaning we do not aim to preserve all mutants killed by the original test case, while our metric MPR does take all mutants killed into account. Therefore, we do not construct test cases that necessarily have high MPR. Furthermore, we also measure the CPR of these reduced test cases, which we do not use as guidance to perform N-mutant test-case reduction.

8. RELATED WORK

Test-case reduction aims to reduce the size or complexity of test cases while preserving particular properties of these test cases. This is essentially a search in the space of possible modifications to the original test case. In most cases, the only modification allowed to the test case is removing or replacing a part of the test case [20, 21, 24].

As the goal of test-case reduction is to speed up testing, it is similar to the many techniques studied in previous projects to speed up regression testing, including regression test selection, test prioritization, and test-suite reduction [23]. The most similar of these techniques to test-case reduction is test-suite reduction. Whereas test-case reduction aims to reduce a single test case, test-suite reduction aims to reduce the size of an entire test suite while preserving some measure of quality (typically by some metric of fault-detection capability) for the reduced test suite. Many studies in the past have investigated different test-suite reduction algorithms [10, 11] or how effective test-suite reduction techniques are [17, 19]. We imagine test-case reduction and test-suite reduction can be easily combined, where one can first reduce a test suite to remove any redundant test cases, and then from the remaining test case in the reduced test suite one can perform test-case reduction to speed up the testing process even more.

Delta debugging [24] is the most common technique for reducing the size of test cases. Given a property of interest and a test case, the algorithm reduces the test case to one that preserves the property, such that no single part can be removed without losing the property. Cause reduction [7] is a generalization of delta-debugging, where a test case is reduced until removing any one part causes the reduced test case to not cover everything the original test case covers. Our non-adequate test-case reduction techniques are a further generalization of cause reduction, where we relax the constraint of needing to cover everything the original test case covers and instead accept a reduced test case that covers at least some percentage.

Mutants of a program are variants of the program with a small syntactic change from the original program. Studies suggest that mutants can be used as proxies for real software faults [2, 13], so if a test suite or an individual test case can kill more mutants, it can potentially detect more real bugs. Moreover, mutants subsume a large number of structural coverage metrics. Thus, it is reasonable to reduce test cases with respect to mutants.

9. CONCLUSION

Having smaller test cases is desirable for a developer to be able to run test cases faster and to comprehend them easier for debugging. Test-case reduction is one approach to reduce the size of test cases. Previous research has studied how to conduct test-case reduction such that the reduced test cases completely preserve some property of the original test case. We have introduced a more general approach to test-case reduction, called non-adequate test-case reduction, that does not force the reduced test case to completely preserve some property of the original test case but rather only partially preserve the property. Specifically, we introduce two reduction techniques, called C%-coverage and N-mutant, that take this non-adequate test-case reduction approach. The results show that these techniques can substantially reduce the size of the test cases while still preserving a large percentage of all coverage of mutants killed by the original test cases. We believe that non-adequate test-case reduction offers new and exciting opportunities to improve testing in particular and software development in general.

10. REFERENCES
Figure 13: MPR vs. SRR of C%-coverage test-case reduction – SpiderMonkey

Figure 14: MPR vs. SRR of C%-coverage test-case reduction – YAFFS2

Figure 15: MPR vs. SRR of C%-coverage test-case reduction – Gzip

Figure 16: MPR vs. SRR of C%-coverage test-case reduction – Grep

Figure 17: CPR vs. SRR of C%-coverage test-case reduction – SpiderMonkey

Figure 18: CPR vs. SRR of C%-coverage test-case reduction – YAFFS2

Figure 19: CPR vs. SRR of C%-coverage test-case reduction – Gzip

Figure 20: CPR vs. SRR of C%-coverage test-case reduction – Grep
Figure 21: MPR vs. SRR of N-mutant test-case reduction – SpiderMonkey

Figure 22: MPR vs. SRR of N-mutant test-case reduction – YAFFS2

Figure 23: MPR vs. SRR of N-mutant test-case reduction – Gzip

Figure 24: MPR vs. SRR of N-mutant test-case reduction – Grep

Figure 25: CPR vs. SRR of N-mutant test-case reduction – SpiderMonkey

Figure 26: CPR vs. SRR of N-mutant test-case reduction – YAFFS2

Figure 27: CPR vs. SRR of N-mutant test-case reduction – Gzip

Figure 28: CPR vs. SRR of N-mutant test-case reduction – Grep
Figure 29: MPR vs. CPR of $N$-mutant test-case reduction

Figure 30: MPR vs. CPR of $C\%$-coverage test-case reduction


